Assessing Test Set Adequacy for Object-Oriented Programs Using Class Mutation

Sun-Woo Kim, John A. Clark, John A. McDermid

Department of Computer Science
The University of York
York, YO10 5DD, England.
E-Mail: {sunwoo, jac, jam}@cs.york.ac.uk

Abstract. The object-oriented paradigm has seen widespread acceptance by the software development community but the testing of programs written in object oriented languages is less well developed than the testing of programs written in conventional ones. This paper introduces Class Mutation which is a form of OO-directed selective mutation testing that provides a means of assessing how good developed test sets are for OO programs. Experimental results are given for the application of three particular mutation operators for the Java language to assess test sets developed to satisfy a conventional criterion.

Keywords: Object-oriented, Java, mutation testing, testing.

1 Introduction

Software testing is one of the main processes in software development which attempts to reveal the presence of errors (Myers 1979). As the majority of recent system developments have been performed under the object-oriented (OO) paradigm – development strategies, methods, and languages – there has been a growing interest in, and need for, object-oriented software testing.

With any software testing a fundamental question arises “How good is my testing?” Test adequacy, which is measured by adequacy criteria/axioms, indicates whether or not a program has been tested enough for it to be reasonable to stop testing. Although it is a relatively simple matter to determine whether a given test set achieves specific measurable criteria (typically requirements about the degree of statement coverage, branch coverage or data use), it is quite another to demonstrate convincingly that the attainment of such criteria really corresponds to adequate testing of a program. Indeed, the unthinking use of such criteria may actually lead to poor testing, since it encourages criterion satisfaction rather than, for example, effective fault finding. In addition, it is highly doubtful whether satisfying traditional adequacy criteria really corresponds to effectively testing of an OO program.
This paper introduces Class Mutation – a form of selective mutation testing that can be used to assess the adequacy of a test set for a program written in an OO language (Java). Class Mutation targets faults that are likely to occur due to OO unique features such as inheritance, polymorphism, and so on. The rationale for this approach is that the faults influenced by the OO features are a subset of the actual software faults in OO programs, and thus, to be adequate, a test set should be able to demonstrate at least the absence of those identified faults.

Section 2 studies the faults associated with object-oriented features. Section 3 describes the process of Class Mutation and the mutation operators that represent the OO faults identified in Section 2. Section 4 evaluates a traditional adequacy criterion (control-flow path coverage) with Class Mutation, and finally our conclusions and further work are presented in Section 5.

2 A Study of the Faults Influenced by OO Features

The majority of the studies of software faults are based on software developed with the traditional structural paradigm and procedural languages. If the characteristics of software faults are not affected by the object-oriented paradigm, existing fault categories can be applied directly to examine the adequacy of test sets for object-oriented programs. However, it is observed that some differences (the frequency, distribution, and detectability of faults, or even the introduction of new types of faults) appear in object-oriented programs. Purchase (1991) states that there is enormous scope for subtle errors with complex polymorphic relations, and the properties of inheritance and dynamic binding introduce new opportunities for the creation of bugs. Marick (1995) claims that the encapsulation feature prevents many of the problems that result from uncontrolled data scoping. Wilde (1992) and Kung (1994) state that the method is less complex and its size considerably smaller than the module of conventional systems. Thus an OO system consists of numbers of interactions between small components, which implies more opportunities for integration/interface faults. The faults that occur in the interfaces between objects and classes are aggravated by reuse and inheritance which give rise to distributed class descriptions.

This section extracts some examples of possible fault scenarios from the literature and discusses their implications for testing and test adequacy. Initially, three features – abstract data types, polymorphism, and inheritance are selected for the study.

2.1 Faults in the Use of Abstract Data Types

The main difference in data types between procedural and OO programs is the use of an abstract data type (ADT), which consists of data attributes and a set of operations that can be applied to instances of the type. It is observed that faults often arise in the usage of ADTs. For example, derived class objects can be manipulated through pointers or references to base class objects. Such pointers and references are
said to behave polymorphically (as if they had multiple types) and often become the source of subtle errors (Meyers 1992).

A C++ example, provided by Meyers (1992), shows an error in vertical polymorphism. Suppose a programmer uses a pointer to the base class (EnemyTarget) to store the object belonging to a derived class (EnemyTank).

```cpp
class EnemyTarget { ... }
class EnemyTank: public EnemyTarget { ... }

EnemyTarget *targetPtr = new EnemyTank;
... delete targetPtr;
```

An error arises when you get rid of the object by calling `delete` because the EnemyTarget destructor turns out to be nonvirtual. The compiler only calls the base class’s destructor instead of the destructor of the derived class although the pointer really points to an object of the derived class. As a result, counting the number of EnemyTank objects in your application will be incorrect.

The following example (Chen 1995) shows a fault that can arise in downcasting (casting down the inheritance hierarchy).

```cpp
class Stack: public Object { // inherits Object
    public : void push (Object*);
    Object* pop();
    ...
};
class X: public Object {...}; // inherits Object
class Y: public Object {...}; // inherits Object

Y* Yobj  = new Y;
Stack Ystack;
Ystack.push(Yobj);
X* topThing = (X*)Ystack.pop();
```

An error occurs because the object returned by the method `pop` is swapped to the incompatible type. That is, the `Object*` returned from `Ystack.pop()` is actually a type `Y`, but cast to `X`, which will result in a type error. This kind of error can arise because it is possible to downcast `Object` (supertype) to `X` (subtype), but ‘`Y` is not `X`’ although ‘`Y` is `Object`’ and ‘`X` is `Object`’.

A mixed use of compatible types frequently appears in object-oriented programs, causing type inconsistency. It is claimed that it is almost impossible to create an efficient and flexible system without doing so, even though it is generally incorrect to logically or arithmetically combine objects whose types are different (Beizer 1990).

### 2.2 Faults in Implementation Reuse

Inheritance allows a class to directly use the data representation and methods defined by a superclass. A subclass may simply extend a base class, it may need to implement the methods of an abstract base class, or it may hide some member
variables and/or override the methods of a super class. In any case, it is important to fully understand the properties of a parent class and their possible interactions with properties in a subclass when creating a subclass. Errors are often caused by failing to do this.

The following example, described by Marick (1995), highlights the problems that may arise in inheritance. set_desired_temperature of the class refrigerator allows the temperature to be between 5 °C and 20 °C. Calibrate puts the actual refrigerator through various cooling cycles and uses sensor readings to calibrate the cooling unit.

```cpp
class refrigerator {
public:
    void set_desired_temperature(int temp);
    int get_temperature();
    void calibrate();
private:
    int temperature;
};
```

A more capable model, better_refrigerator, was then derived from the refrigerator. Since it can cool to -5 °C, set_desired_temperature has been reimplemented, but calibrate was unchanged. Suppose that calibrate works, in part, by dividing sensor reading by temperature. The temperature can be zero in better_refrigerator, and it will cause a divide-by-zero failure that would not appear in refrigerator.

This example demonstrates the adequacy criterion for inheritance addressed by Perry and Kaiser (1990). When we add a new subclass (or modify an existing subclass), we must retest the methods inherited even though the method is unchanged. It is because an error can arise in the interaction between the inherited features and new/changed features in a subclass. The fault observed in the example occurred due to the conflict between the inherited feature (calibrate) and the changed feature (set_desired_temperature) in the subclass, and it could be detected if calibrate is retested in the context of better_refrigerator. This example also shows a modularity violation that arises in object-oriented programs due to inheritance, consequently causing dependency problems in testing and maintenance.

Now suppose the code below references the class refrigerator. This code uses the temperature as an index into an array. That is legal for an object of class refrigerator because the temperature must be in the range [5,20]. However, &ref may also refer to an object of the derived class, better_refrigerator. If so, the temperature may be in the range [-5,20], which allows an invalid array access. The implication of this example is that handle_refrigerator should be tested against each possible class, not against just one, because &ref can refer to all derived classes of refrigerator due to the feature dynamic binding.

```cpp
handle_refrigerator(refrigerator &ref) {
    extern record Global_array[212];
    int index=ref.get_temperature();
    if (Global_array[index].count >0) ...
}
```
In this section, the fault scenarios have been presented under the two headings – abstract data types and implementation reuse. However, the faults can rarely be isolated to a single cause, and actually arise by the interactions of several related features. For example, type errors usually occur when the use of a class type is inter-related with inheritance and/or polymorphism. Note that some faults may not behave as explained here because each programming language has a different way of implementing object-oriented properties. Errors are sometimes caused by a programmer’s misconception of subtle differences in features between languages.

3 Mutation Technique and Class Mutation

Mutation testing (DeMillo et al. 1979) assesses test set quality by examining whether a test set is able to differentiate the program under test from small syntactic variations of the original program (mutants). The mutants are made by applying a set of predefined transformation rules called mutation operators. A test set is executed over the mutants and the output of mutant execution is compared against the original program. If the output of a mutant is different from the original program, the mutant is said to be distinguished (killed) by the test set. This case is satisfactory because we assume the test cases which are good at distinguishing injected faults ought also to be good at detecting real faults. The mutants which produce the same output as the original program are called live mutants. A mutant may remain live because either it is an equivalent mutant or the test set is inadequate to kill the mutant. If the mutant is an equivalent mutant, it would always produce the same output, hence it cannot be killed. If the test set is inadequate, the test set must be augmented to kill the live mutants. Errors are found during the process of analysing and killing the live mutants. A mutation adequacy score, which is the ratio of dead mutants to non-equivalent mutants, gives a quantitative measurement of the quality and effectiveness of the test set.

Class Mutation applies the mutation technique to OO (Java) programs. The basic unit of testing and mutation is a class since the minimum executable program unit in Java is a class. The operators of Class Mutation are targeted at OO-specific features which Java provides – class declarations and references, single inheritance, interface, information hiding, and polymorphism. In another report (Kim et al. 1999) we show how to derive systematically a set of Java mutation operators. In this paper, experiments are carried out, for illustrative purposes, using three mutation operators designed to simulate faults of the types indicated in Section 2. This section gives a description of each operator. Code examples for each operator are shown in Section 4.

- CRT (Compatible Reference Type) operator – Type replacement

This operator replaces a reference type with all the compatible types (the names of the other classes and interfaces) found from a cluster. Since Java supports only vertical relations (inheritance), the compatible types would be related to each other.

---

1 A cluster is a number of classes related to/interacting with each other by the relationships that can exist in OO programs such as inheritance, class nesting, aggregation, association, etc.
with super/sub relations in an inheritance hierarchy. It is expected that subtle type errors can be highlighted by this mutation.

- **CON (Constructor) operator – Initial states and object replacement**
  A Java class usually provides one or more constructors to capture the different ways of creating objects (constructor overloading). This operator replaces a constructor with other overloaded constructors. In addition, it is possible to replace it with the constructors of subclasses. When the constructor is replaced with the other constructors of the same type, it will simply create an object with different initial states. However, when the constructors of a subclass are used as a mutation, an object of the subtype is actually created. Object initialisation, which is one of the most frequent types of software error, is related to this operator.

- **OVM (Overriding Method) operator - Method replacement**
  This operator generates a mutant by deactivating the overriding method so that a reference to the overriding method actually goes to the overridden method. The rationale behind the OVM operator is that the overriding method in a subclass should have different functionality to the overridden method in a super class. Otherwise, the programmer would simply inherit the method of the super class instead of re-implementing it. If a test set fails to see any difference whether the overriding method is called or the overridden method is called, there may be some semantic errors or opportunities for coincidental correctness.

All the mutant operators described in this section attempt to alter the program semantics through a simple syntactic change. While the kind of error that traditional mutation operators aim at is restricted to a unit level, the operators for the OO mutation method consider the relations in a cluster and get information to make a mutant from the cluster although the basic mutation unit is a class. This is rather an inevitable consequence because the OO faults the mutation operators are aiming to model are actually caused by the relationships between classes.

4 Experimentation with Control-Flow Path Coverage

The examples (Queue, Dequeue, and PriorityQueue classes) were extracted from the ADS class library (Winder 1997). The class Queue represents a ‘first-in-first-out’ structure, and Dequeue (double-ended queue) allows elements to be added to or removed from both head and tail. PriorityQueue is a multi-level Queue with a priority. The default insert is at the lowest priority and the default remove is at the highest priority. The class structure of the example classes and related classes is shown in Fig. 1. A set of test cases and correct output for each class were prepared to meet the control-flow path coverage criterion. That is, the test cases were generated to cover all the paths in the control-flow graph of each class. The control-flow graph of Queue is shown in Fig. 2.
The mutation operators (CRT, CON, and OVM) were applied to Queue, Dequeue and PriorityQueue to generate the mutants. The following tables show some of the mutants of each class and the faults those mutants are intended to represent.

**Table 1.** Mutants of Queue (QM stands for Queue Mutant)

<table>
<thead>
<tr>
<th>Mutants</th>
<th>Fault descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>QM1 (CON)</td>
<td>The method clone of Queue class makes a clone of Dequeue instead of Queue.</td>
</tr>
</tbody>
</table>
QM2 (CON) The method clone of Queue makes a clone of PriorityQueue instead of Queue.

QM3 (CRT) The parameter type of a Queue constructor is widened from Sequence (subinterface) to Container (super interface).

QM4 (CON) The storage created to contain Queue objects is DList (Double linked list) instead of List.

QM5 (CON) The storage created to contain Queue objects is DLLList (Double linked list but different implementation) instead of List.

QM6 (CON) The storage created to contain Queue object is VectorSequence instead of List.

Table 2. Mutants of Dequeue (DM stands for Dequeue Mutant)

<table>
<thead>
<tr>
<th>Mutants</th>
<th>Fault descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>DM1 (CRT)</td>
<td>The object created by the clone method of Dequeue has the declared type of Queue instead of Dequeue.</td>
</tr>
<tr>
<td>DM2 (CON)</td>
<td>The storage created to contain Dequeue objects is List instead of DLLList.</td>
</tr>
<tr>
<td>DM3 (CON)</td>
<td>The storage created to contain Dequeue objects is DList instead of DLLList.</td>
</tr>
<tr>
<td>DM4 (CON)</td>
<td>The storage created to contain Dequeue objects is VectorSequence instead of DLLList.</td>
</tr>
<tr>
<td>DM5 (OVM)</td>
<td>The equals of Dequeue is removed (the condition that checks whether the compared object is an instance of Dequeue is lost).</td>
</tr>
<tr>
<td>DM6 (OVM)</td>
<td>The method clone of Queue is called (it makes a clone object of Queue instead of Dequeue).</td>
</tr>
</tbody>
</table>

Table 3. Mutants of PriorityQueue (PM stands for PriorityQueue Mutant)

<table>
<thead>
<tr>
<th>Mutants</th>
<th>Fault descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>PM1 (CRT)</td>
<td>The object created by clone of PriorityQueue has the declared type Queue instead of PriorityQueue.</td>
</tr>
<tr>
<td>PM2 (CON)</td>
<td>The clone creates an object with a wrong initial state. The default number of levels (10) is given instead of the number of levels specified in the constructor parameter.</td>
</tr>
<tr>
<td>PM3 (CON)</td>
<td>The method addPriorityLevel adds a new priority level of PriorityQueue instead of Queue.</td>
</tr>
<tr>
<td>PM4 (CON)</td>
<td>The method addPriorityLevel adds a new priority level of Dequeue instead of Queue.</td>
</tr>
<tr>
<td>PM5 (CON)</td>
<td>The container of PriorityQueue objects is created with a default size instead of the initial size given by users.</td>
</tr>
<tr>
<td>PM6 (CON)</td>
<td>The created container of PriorityQueue is a List instead of VectorSequence.</td>
</tr>
<tr>
<td>PM7 (CON)</td>
<td>The created container of PriorityQueue is a DLLList instead of VectorSequence.</td>
</tr>
</tbody>
</table>
| PM8 (OVM) | The method makeEmpty of Queue is called. Only the first level
of storage is emptied (for loop that repeats emptying the storage until the end of the priority levels is lost).

| PM9 (OVM) | The method `isEmpty` that overrides that of `Queue` is removed. Only the first level is checked for emptiness. |
| PM10 (OVM) | The method `size` of `Queue` is invoked instead of that of `PriorityQueue`. It does not count all the levels of containers. |
| PM11 (OVM) | The `equals` of `Queue` is called to compare equality of `PriorityQueue` objects. It neither checks whether the object is an instance of `PriorityQueue` nor compares all the levels of containers. |

Code examples below show the original code that is the subject of mutation, together with corresponding mutants (QM1 in Table 1, DM1 and DM5 in Table 2).

- A Dequeue Mutant (DM1) by CRT Operator
  The `clone` method of `Dequeue` makes a clone of the `Dequeue` object. A type fault is injected by changing the declared type of object `q` in the method `clone`, from `Dequeue` to `Queue`.

  The original code (the `clone` method of `Dequeue` class is):
  ```java
  public synchronized Object clone() {
      Dequeue q = new Dequeue();
      //The declared type is Dequeue.
      q.storage = (Sequence)storage.clone() ;
      return q ;
  }
  ```

  The mutant DM1 is:
  ```java
  public synchronized Object clone() {
      Queue q = new Dequeue() ;
      //The declared type is replaced with Queue.
      q.storage = (Sequence)storage.clone() ;
      return q ;
  }
  ```

- A Queue Mutant (QM1) by CON operator
  The `clone` method of `Queue` is intended to make a clone of `Queue`. So, changing the constructor `Queue()` to `Dequeue()` is a fault – a wrong type of object is created.

  The original code (the `clone` method of `Queue` class is):
  ```java
  public synchronized Object clone() {
      Queue q = new Queue () ;
      //The constructor is Queue()
      q.storage = (Sequence)storage.clone() ;
      return q ;
  }
  ```

  The mutant QM1 is:
public synchronized Object clone() {
    Queue q = new Dequeue();
    // The constructor is replaced with Dequeue()
    q.storage = (Sequence)storage.clone();
    return q;
}

• A Dequeue Mutant (DM5) by OVM operator

The method equals in Dequeue overrides that of Queue because it needs to check whether the compared object is an instance of Dequeue while the equals of Queue checks if the object is an instance of Queue. When the equals of Dequeue is deactivated by the OVM operator, the equals of Queue will be invoked instead, which checks if the object is an instance of Queue, not Dequeue – a missing condition error.

The equals method in Dequeue class is deactivated by commenting out,

```java
/*public synchronized boolean equals(final Object o){
    if ((o == null) || ! (o instanceof Dequeue))
        return false;
    return super.equals(o);
}*/
```

and the following code in class Queue is invoked instead.

```java
public synchronized boolean equals(final Object o) {
    if (o == null || ! (o instanceof Queue))
        return false;
    synchronized (o) {
        return storage.equals(((Queue)o).storage);
    }
}
```

Of course the programmer(s) of this example did not make an error – the required condition is included by reimplementing the method of superclass. One might wonder what is achieved by distorting the already-correct code. Remember that the main aim of mutation testing is adequacy measurement of a test set. If a current test set cannot kill this mutant (i.e., it cannot distinguish the omission of the method equals of Dequeue), that test set is certainly not good enough to uncover the situation that the programmer actually forgot to reimplement the equals method to check for equality only between Dequeue objects.

Table 4 shows the result of the mutation execution for each class. In the Queue mutation, most of the CON mutants (QM1, 4, 5, and 6) remain live. These live mutants could be killed by adding more test cases. The mutation adequacy score of Queue is 33.33%. The only live mutant in the Dequeue mutation is DM1. DM1 turns out to be an equivalent mutant, so no test case could kill it. The mutation adequacy score of Dequeue is 100% - i.e., the test set was able to kill all the killable mutants. PriorityQueue has 4 live mutants. PM1 cannot be killed since it is an equivalent mutant.

---

2 The mutation adequacy score is the ratio of dead mutants to the total number of killable (non-equivalent) mutants.
mutant, but PM5, 6, and 7 are all killable mutants. The mutation adequacy score of PriorityQueue is 70%.

Table 4. Results of mutation execution

<table>
<thead>
<tr>
<th>Class names</th>
<th>Killed mutants</th>
<th>Live mutants</th>
<th>Mutation score</th>
</tr>
</thead>
<tbody>
<tr>
<td>Queue</td>
<td>QM2, QM3</td>
<td>QM1, QM4, M5, QM6</td>
<td>33.33%</td>
</tr>
<tr>
<td>Dequeue</td>
<td>DM2, DM3, DM4, DM5, DM6</td>
<td>DM1</td>
<td>100%</td>
</tr>
<tr>
<td>PriorityQueue</td>
<td>PM2, PM3, PM4, PM8, PM9, PM10, PM11</td>
<td>PM1, PM5, PM6, PM7</td>
<td>70%</td>
</tr>
</tbody>
</table>

Table 5 shows the mutation score for each mutation operator type of the mutants in Queue, Dequeue, and PriorityQueue. All the CRT mutants are killed, but this result is not very credible since there are only 3 CRT mutants and two of them are equivalent mutants. All the mutants of the OVM type are also killed, but only 50% of the mutants by the CON operator are killed.

Table 5. Mutation adequacy score for each mutation operator type

<table>
<thead>
<tr>
<th>Types</th>
<th>Killed mutants</th>
<th>Live mutants</th>
<th>Mutation score</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRT</td>
<td>QM3</td>
<td>DM1, PM1 (Both are equivalent mutants.)</td>
<td>100%</td>
</tr>
<tr>
<td>CON</td>
<td>QM2, DM2, DM3, DM4, PM2, PM3, PM4</td>
<td>QM1, QM4, QM5, QM6, PM5, PM6, PM7</td>
<td>50%</td>
</tr>
<tr>
<td>OVM</td>
<td>DM5, DM6, PM8, PM9, PM10, PM11</td>
<td>PM11</td>
<td>100%</td>
</tr>
</tbody>
</table>

The result of mutation execution shows that even though the test set giving path coverage was able to kill many mutants, some of them remain live. The test set was not effective at uncovering the faults introduced by the CON operator. The CON operator addresses the issue of ‘subtyping’ which may not show a significant difference in functionality in many cases. It is likely that a system works as intended when an object of a subtype is inadvertently running instead of that of a super type. Thus, testing may fail to distinguish any difference unless it executes the behaviours that belong to only the subtype, or explicitly compares the object types.

5 Conclusion

The creation of an effective test set is a crucial element of the testing process and many testing strategies have been advocated. Where the testing of programs written in an OO language is considered, methodological guidance is thinner (since the subject is relatively new) but it is increasing. In all cases, the correspondence between tests created using the recommended strategies and effectiveness in terms of fault finding
ability is not cut and dried. Many methods are motivated by experience and so one would expect some positive correlation, but the tester is still left with uncertainty as to how well the program under consideration really has been exercised.

Class Mutation provides one means by which the effectiveness of a test set can be determined. In this paper, the results of small-scale experimentation have been presented based on only a few mutation operators (targeted at OO-specific features). These have shown how test sets produced with a conventional criterion (here control-flow) may be lacking. As well as indicating how well a particular program has been tested, program mutation looks promising as a candidate means for comparing the effectiveness of different test creation strategies (though much greater-scale experimentation is required to confirm this).

In future work, Class Mutation will be extended to implement more mutation operators. Other object-oriented features such as access control, exception handling, etc. will be examined to identify the plausible faults in OO programs. In addition, some efficiency issues in the mutation method will be addressed. For example, Class Mutation currently deals only with strong mutation, but we intend to add a weak mutation facility to look at the “inside” of class states.

References

Meyers S.: Effective C++: 50 Specific Ways to Improve Your Programs and Designs, Addison-Wesley (1992)